IA014: Advanced Functional Programming

4. Polymorphism and Type Inference

Jan Obdržálek obdrzalek@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Polymorphism I

Motivation

Typing id $\equiv \lambda x.\ x$

- In $\lambda^{\rightarrow}$:

$$(\lambda x : \mathrm{Nat}.\ x) : \mathrm{Nat} \rightarrow \mathrm{Nat}$$
$$(\lambda x : \mathrm{Bool}.\ x) : \mathrm{Bool} \rightarrow \mathrm{Bool}$$
$$\ldots$$

- What we really mean:

$$(\lambda x.\ x) : \forall \alpha . \alpha \rightarrow \alpha$$

- In HASKELL:

```
Prelude> :info id
id :: a -> a   -- Defined in ‘GHC.Base’
```

More examples

$$\text{double} := \lambda f.\lambda x.f(f\ x) : \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$$

Lists

- $\text{null} : \forall\alpha.[\alpha] \to \text{Bool}$
- $\text{nil} : \forall\alpha.[\alpha]$
- $\text{cons} : \forall\alpha.\alpha \to ([\alpha] \to [\alpha])$
- $\text{hd} : \forall\alpha.[\alpha] \to \alpha$
- $\text{tl} : \forall\alpha.[\alpha] \to [\alpha]$

Polymorphism

The id $\equiv \lambda x.\ x$ function is, by its nature, *polymorphic*.

Types of polymorphism

- *parametric* polymorphism
  - "all types"
  - Allows single piece of code to be typed parametrically, i.e. using type variables, and instantiated when needed.
  - All instances behave the same.
- *ad-hoc* polymorphism
  - "some types"
  - *overloading:* one function has many implementations (differeng by the types of the arguments)
  - May behave differently for different types of arguments.

Goal: extend λ-calculus with parametric polymorphism

so we can use functions like id

Extending λ – syntax

System HM (Hindley-Milner)

term and values

$\mathrm{t} ::=$	x	variable
$\mid$	$\mathrm{t}\ \mathrm{t}'$	application
$\mid$	$\lambda x.\mathrm{t}$	abstraction
$\mid$	$\mathrm{let}\ \ x = \mathrm{t}\ \ \mathrm{in}\ \ \mathrm{t}$	let binding

monotypes

$\mathrm{T} ::=$	α	type *variable*
$\mid$	$\mathrm{T} \to \mathrm{T}$	function *type*

type schemes (polytypes)

$\mathrm{S} ::=$	T	monotype
$\mid$	$\forall\vec{\alpha}.\mathrm{T}$	generic type

Type variables and schemes

$$\mathrm{T} ::= \alpha \mid \mathrm{T} \to \mathrm{T} \qquad \mathrm{S} ::= \mathrm{T} \mid \forall \vec{\alpha}.\mathrm{T}$$

Type variables

- $\alpha, \alpha', \beta \ldots$
- can stand for any *monotype*
- we assume we have an infinite supply of type variables

Type schemes

- either a monotype T or a *generic type* $\forall \alpha_1 \ldots \forall \alpha_n.\mathrm{T}$
- $\alpha_1, \ldots, \alpha_n$ are *generic* type variables
- notions of *free/bound* type variables (notation $FV(\mathrm{S})$)
- can be *instantiated*

Type substitution/instantiation

substitution

- mapping from type variables to types
- notation: $\theta = \{\mathrm{T}_1/\alpha_1, \ldots, \mathrm{T}_n/\alpha_n\}$
- $\theta\mathrm{S}$ – application to a type scheme S
    - replace each *free occurrence* of α_i in S with T_i
    - rename generic variables if necessary
    - θS is an *instance* of S
- naturally extends to contexts:
  $\theta(\Gamma) = \theta(x_1 : \mathrm{T}_1, \ldots, x_k : \mathrm{T}_k) = x_1 : \theta\mathrm{T}_1, \ldots, x_k : \theta\mathrm{T}_k$
- can be composed: $\theta_2\theta_1 S$

Type ordering

What is the relationship among the many types of $\mathrm{id} \equiv \lambda x.x$?

$$\forall\alpha.\alpha \to \alpha \qquad \mathrm{Nat} \to \mathrm{Nat} \qquad (\mathrm{Nat} \to \mathrm{Nat}) \to (\mathrm{Nat} \to \mathrm{Nat})$$

- $\forall\alpha.\alpha \to \alpha$ is more *general* than any of the others
- $\mathrm{Nat} \to \mathrm{Nat}$ is *more specialized* than $\forall\alpha.\alpha \to \alpha$
- we write $\forall\alpha.\alpha \to \alpha \sqsubseteq \mathrm{Nat} \to \mathrm{Nat}$

Specialization rule

$$\frac{\mathrm{T}' = \{\mathrm{T}_i/\alpha_i\}\mathrm{T} \qquad \beta_i \notin FV(\forall\alpha_1 \ldots \forall\alpha_n.\mathrm{T})}{\forall\alpha_1 \ldots \forall\alpha_n.\mathrm{T} \sqsubseteq \forall\beta_1 \ldots \forall\beta_m.\mathrm{T}'}$$

- $\forall\beta_1 \ldots \forall\beta_m.\mathrm{T}'$ is an *instance* of $\forall\alpha_1 \ldots \forall\alpha_n.\mathrm{T}$

Type inference

Motivation

Two flavours of $\lambda^{\rightarrow}$

- Implicitly typed
  - Haskell B. Curry, 1934
  - $\boldsymbol{I} = (\lambda x.x) : A \rightarrow A$
  - $\boldsymbol{I} = (\lambda x.x) : (A \rightarrow B) \rightarrow (A \rightarrow B)$
- Explicitly typed
  - Alonzo Church, 1940
  - $\boldsymbol{I}_A = (\lambda x : A.x) : A \rightarrow A$
  - $\boldsymbol{I}_{A \rightarrow B} = (\lambda x : (A \rightarrow B).x) : (A \rightarrow B) \rightarrow (A \rightarrow B)$

Goal: Support implicit typing

To do this, we must be able to automatically infer (reconstruct) types of terms.

Type inference (reconstruction)

- The goal is to *automatically derive* types for the term.
- At the heart of e.g. ML and HASKELL.
- Also used for *type-checking*.
  Some of the types may be given explicitly.

Hindley-Milner algorithm

- first discovered by Hindley (1969)
- in the context of ML rediscovered by Milner (1978)
- formal analysis and proofs: Milner and Damas (1982)
- also called Damas-Hindley-Milner algorithm
- works for lambda calculus with `let`-polymorphism

Type inference idea

What is the type of $\lambda x.\, \mathrm{succ}\, x$?

- Assume $\Gamma = \mathrm{succ} : \mathrm{Nat} \rightarrow \mathrm{Nat}$
- We do not know the type of x (yet) so we put $x : \alpha$
- From environment we know that $\mathrm{succ} : \mathrm{Nat} \rightarrow \mathrm{Nat}$
- For succ to be applied to x we need x to be of type Nat
- Solution: substitution $\theta = \{\mathrm{Nat}/\alpha\}$.

What about $\mathrm{let}\ \ \mathrm{id} = \lambda x.x\ \ \mathrm{in}\ \ (\mathrm{id}\, \mathrm{false}, \mathrm{id}\, 0)$?

- We do not know the type of x (yet) so we put $x : \alpha$
- Therefore $\mathrm{id} : \alpha \rightarrow \alpha$
- On `let`-binding we generalize this to $\forall \alpha : \alpha \rightarrow \alpha$
- For each use of id we instantiate α with a fresh variable say $\beta \rightarrow \beta$ in the first case and $\gamma \rightarrow \gamma$ in the second
- β gets unified with Bool and γ with Nat
- the sought-for type is therefore $(\mathrm{Bool}, \mathrm{Nat})$.

System HM – Typing rules

$$\frac{x : \mathrm{S} \in \Gamma}{\Gamma \vdash x : \mathrm{S}} \text{ (T-Var)}$$

$$\frac{\Gamma, x : \mathrm{T}_1 \vdash \mathrm{t} : \mathrm{T}_2}{\Gamma \vdash \lambda x.\mathrm{t} : \mathrm{T}_1 \to \mathrm{T}_2} \text{ (T-Abs)}$$

$$\frac{\Gamma \vdash \mathrm{t}_1 : \mathrm{T}_1 \to \mathrm{T}_2 \qquad \Gamma \vdash \mathrm{t}_2 : \mathrm{T}_1}{\Gamma \vdash \mathrm{t}_1\ \mathrm{t}_2 : \mathrm{T}_2} \text{ (T-App)}$$

$$\frac{\Gamma \vdash \mathrm{t}_1 : \mathrm{S} \qquad \Gamma, x : \mathrm{S} \vdash \mathrm{t}_2 : \mathrm{T}}{\Gamma \vdash \mathrm{let}\ \ x = \mathrm{t}_1\ \ \mathrm{in}\ \ \mathrm{t}_2 : \mathrm{T}} \text{ (T-Let)}$$

$$\frac{\Gamma \vdash \mathrm{t} : \mathrm{S}' \qquad \mathrm{S}' \sqsubseteq \mathrm{S}}{\Gamma \vdash \mathrm{t} : \mathrm{S}} \text{ (T-Inst)}$$

$$\frac{\Gamma \vdash \mathrm{t} : \mathrm{S} \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash \mathrm{t} : \forall \alpha.\mathrm{S}} \text{ (T-Gen)}$$

Observation

- The first four rules are syntax driven.
- We have a choice for T-Inst and T-Gen.

`Let` polymorphism

T-Abs vs T-Let

- no type can be inferred for $\lambda f.(f\,\mathrm{true}, f\ 0)$
- type of $\mathrm{let}\ \ f = \lambda x.x\ \ \mathrm{in}\ \ (f\,\mathrm{true}, f\ 0)$ is $(\mathrm{Bool}, \mathrm{Nat})$

The rules on previous slide give so-called *let polymorphism*

- type of *parametric polymorphism*
- let-expressions allow local bindings to have polymorphic types

  that's why `let` is in the syntax!
- λ-bound variables are always assumed to be monotypes
- `let`-bound variables can have polymorphic types, because *we know what they are bound to*
- strikes balance between expressivity and decidability

Type inference examples

(1) Show $\Gamma \vdash \mathrm{id}\ n : \mathrm{Nat}$ for $\Gamma = \mathrm{id} : \forall\alpha.\alpha \to \alpha, n : \mathrm{Nat}$

$$\dfrac{\dfrac{\dfrac{\mathrm{id} : \forall\alpha.\alpha \to \alpha \in \Gamma}{\Gamma \vdash \mathrm{id} : \forall\alpha.\alpha \to \alpha}\ \text{(T-Var)} \qquad \forall\alpha.\alpha \to \alpha \sqsubseteq \mathrm{Nat} \to \mathrm{Nat}}{\Gamma \vdash \mathrm{id} : \mathrm{Nat} \to \mathrm{Nat}}\ \text{(T-Inst)} \qquad \dfrac{n : \mathrm{Nat} \in \Gamma}{\Gamma \vdash n : \mathrm{Nat}}\ \text{(T-Var)}}{\Gamma \vdash \mathrm{id}\ n : \mathrm{Nat}}\ \text{(T-App)}$$

(2) Show $\mathrm{let}\ \ \mathrm{id} = \lambda x.x\ \ \mathrm{in}\ \ \mathrm{id} : \forall\alpha.\alpha \to \alpha$

$$\dfrac{\dfrac{\dfrac{\dfrac{x : \alpha \in \{x : \alpha\}}{x : \alpha \vdash x : \alpha}\ \text{(T-Var)}}{\vdash \lambda x.x : \alpha \to \alpha}\ \text{(T-Abs)}}{\vdash \lambda x.x : \forall\alpha.\alpha \to \alpha}\ \text{(T-Gen)} \qquad \dfrac{\mathrm{id} : \forall\alpha.\alpha \to \alpha \in \{\mathrm{id} : \forall\alpha.\alpha \to \alpha\}}{\mathrm{id} : \forall\alpha.\alpha \to \alpha \vdash \mathrm{id} : \forall\alpha.\alpha \to \alpha}\ \text{(T-Var)}}{\mathrm{let}\ \ \mathrm{id} = \lambda x.x\ \ \mathrm{in}\ \ \mathrm{id} : \forall\alpha.\alpha \to \alpha}\ \text{(T-Let)}$$

Combining rules

To get an algorithm, it would be nice if the type system is completely syntax directed.

T-Inst

We can instantiate when we introduce a variable:

$$\frac{x : \mathrm{S} \in \Gamma \qquad \mathrm{S} \sqsubseteq \mathrm{S}'}{\Gamma \vdash x : \mathrm{S'}} \text{ (T-Var')}$$

T-Gen

We can generalize immediately at the level of let expressions:

$$\frac{\Gamma \vdash \mathrm{t}_1 : \mathrm{S} \qquad \Gamma, x : \forall \vec{\alpha}.\mathrm{S} \vdash \mathrm{t}_2 : \mathrm{T}}{\Gamma \vdash \mathrm{let}\ \ x = \mathrm{t}_1\ \ \mathrm{in}\ \ \mathrm{t}_2 : \mathrm{T}} \text{ (T-Let')}$$

where $\vec{\alpha} = FV(\mathrm{S}) \setminus FV(\Gamma)$ (generalizing *as far as possible*)

Algorithm W

Algorithm W

INPUT: a context Γ and a term t

OUTPUT: substitution θ and a type T, such that $\theta\Gamma \vdash t : T$

- We will follow [Damas, Milner 1982].
- Along the rules of the type system.
- Uses the *unification algorithm* of Robinson (1965).

Unification

INPUT: types T_1 and T_2
OUTPUT: substitution θ (called *unifier*) such that $\theta T_1 = \theta T_2$
or `fail` if such θ does not exist

$$
\begin{array}{ll}
\texttt{unify}\ \alpha\ T_2 = & \texttt{if}\ \alpha \notin FV(T_2)\ \texttt{then}\ \{T_2/\alpha\}\ \texttt{else fail} \\
\texttt{unify}\ T_1\ \alpha = & \texttt{if}\ \alpha \notin FV(T_1)\ \texttt{then}\ \{T_1/\alpha\}\ \texttt{else fail} \\
\texttt{unify}\ S_1 \to S_1'\ \ S_2 \to S_2' = & \texttt{let}\ \theta_1 = \texttt{unify}\ S_1\ S_2 \\
 & \quad\ \ \theta_2 = \texttt{unify}\ \theta_1(S_1')\ \theta_1(S_2') \\
 & \texttt{in}\ \theta_2\theta_1 \\
\texttt{unify}\ \_\ \_ = \texttt{fail} &
\end{array}
$$

Theorem (Robinson)

Let $\theta = \texttt{unify}(T_1, T_2)$ and κ another unifier of T_1 and T_2. Then there exist θ' such that $\kappa = \theta'\theta$.

In other words, θ is the *most general unifier*.

Algorithm W (1)

$$\frac{x : \mathrm{S} \in \Gamma \qquad \mathrm{S} \sqsubseteq \mathrm{S}'}{\Gamma \vdash x : \mathrm{S}'} \text{ (T-Var')}$$

$$\mathcal{W}(\Gamma, x) = \qquad (\theta_{id}, inst(\mathrm{T})) \qquad \text{where } x : \mathrm{T} \in \Gamma$$

$$inst(\forall \alpha_1 \ldots \alpha_n.\mathrm{T}) = \{\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n\}\mathrm{T} \text{ where } \beta_1, \ldots, \beta_n \text{ are fresh}$$

$$\frac{\Gamma \vdash \mathrm{t}_1 : \mathrm{T}_1 \to \mathrm{T}_2 \qquad \Gamma \vdash \mathrm{t}_2 : \mathrm{T}_1}{\Gamma \vdash \mathrm{t}_1\ \mathrm{t}_2 : \mathrm{T}_2} \text{ (T-App)}$$

$$\begin{aligned}
\mathcal{W}(\Gamma, \mathrm{t}_1\ \mathrm{t}_2) = \quad & \text{let} \quad (\theta_1, \mathrm{T}_1) = \mathcal{W}(\Gamma, \mathrm{t}_1) \\
& \qquad\ (\theta_2, \mathrm{T}_2) = \mathcal{W}(\theta_1\Gamma, \mathrm{t}_2) \\
& \qquad\ \theta_3 = \text{unify}(\theta_2\mathrm{T}_1, \mathrm{T}_2 \to \beta) \text{ where } \beta \text{ is fresh} \\
& \text{in} \quad\ (\theta_3\theta_2\theta_1, \theta_3\beta)
\end{aligned}$$

Algorithm W (2)

$$\frac{\Gamma, x : \mathrm{T}_1 \vdash \mathrm{t} : \mathrm{T}_2}{\Gamma \vdash \lambda x.\mathrm{t} : \mathrm{T}_1 \to \mathrm{T}_2} \text{ (T-Abs)}$$

$$\begin{array}{lll} \mathcal{W}(\Gamma, \lambda x.\mathrm{t}) = & \textsf{let} & (\theta, \mathrm{T}) = \mathcal{W}(\Gamma \cup \{x : \beta\}, \mathrm{t}) \text{ where } \beta \text{ is fresh} \\ & \textsf{in} & (\theta, \theta(\beta \to \mathrm{T})) \end{array}$$

$$\frac{\Gamma \vdash \mathrm{t}_1 : \mathrm{S} \qquad \Gamma, x : \forall \vec{\alpha}.\mathrm{S} \vdash \mathrm{t}_2 : \mathrm{T}}{\Gamma \vdash \mathrm{let}\ \ x = \mathrm{t}_1\ \ \mathrm{in}\ \ \mathrm{t}_2 : \mathrm{T}} \text{ (T-Let)}$$

$$\begin{array}{lll} \mathcal{W}(\Gamma, \mathrm{let}\ \ x = \mathrm{t}_1\ \ \mathrm{in}\ \ \mathrm{t}_2) = & \textsf{let} & (\theta_1, T_1) = \mathcal{W}(\Gamma, \mathrm{t}_1) \\ & & (\theta_2, \mathrm{T}_2) = \mathcal{W}(\theta_1 \Gamma \cup \{x : gen(\theta_1 \Gamma, \mathrm{T}_1)\}, \mathrm{t}_2) \\ & \textsf{in} & (\theta_2 \theta_1, \mathrm{T}_2) \end{array}$$

$$gen(\Gamma, \mathrm{T}) = \forall \vec{\alpha}.\mathrm{T} \ \text{ where } \ \vec{\alpha} = FV(\mathrm{T}) \setminus FV(\Gamma)$$

Algorithm W (complete)

$$\mathcal{W}(\Gamma, x) = \quad (\theta_{id}, inst(\mathrm{T})) \quad \text{where } x : \mathrm{T} \in \Gamma$$

$$\begin{aligned}\mathcal{W}(\Gamma, \mathrm{t}_1\ \mathrm{t}_2) = \quad & \text{let} \quad (\theta_1, \mathrm{T}_1) = \mathcal{W}(\Gamma, \mathrm{t}_1) \\ & \qquad\ (\theta_2, \mathrm{T}_2) = \mathcal{W}(\theta_1\Gamma, \mathrm{t}_2) \\ & \qquad\ \theta_3 = \text{unify}(\theta_2\mathrm{T}_1, \mathrm{T}_2 \rightarrow \beta) \text{ where } \beta \text{ is fresh} \\ & \text{in} \quad (\theta_3\theta_2\theta_1, \theta_3\beta)\end{aligned}$$

$$\begin{aligned}\mathcal{W}(\Gamma, \lambda x.\mathrm{t}) = \quad & \text{let} \quad (\theta, \mathrm{T}) = \mathcal{W}(\Gamma \cup \{x : \beta\}, \mathrm{t}) \text{ where } \beta \text{ is fresh} \\ & \text{in} \quad (\theta, \theta(\beta \rightarrow \mathrm{T}))\end{aligned}$$

$$\begin{aligned}\mathcal{W}(\Gamma, \mathrm{let}\ \ x = \mathrm{t}_1\ \ \mathrm{in}\ \ \mathrm{t}_2) = \quad & \text{let} \quad (\theta_1, T_1) = \mathcal{W}(\Gamma, \mathrm{t}_1) \\ & \qquad\ (\theta_2, \mathrm{T}_2) = \mathcal{W}(\theta_1\Gamma \cup \{x : gen(\theta_1\Gamma, \mathrm{T}_1)\}, \mathrm{t}_2) \\ & \text{in} \quad (\theta_2\theta_1, \mathrm{T}_2)\end{aligned}$$

$$gen(\Gamma, \mathrm{T}) = \forall\vec{\alpha}.\mathrm{T} \ \text{ where } \ \vec{\alpha} = FV(\mathrm{T}) \setminus FV(\Gamma)$$

$$inst(\forall\alpha_1 \ldots \alpha_n.\mathrm{T}) = \{\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n\}\mathrm{T} \ \text{ where } \ \beta_1, \ldots.\beta_n \text{ are fresh}$$

Algorithm W – properties

Theorem (Soundness of $\mathcal{W}$)

If $\mathcal{W}(\Gamma, \mathrm{t}) = (\theta, \mathrm{T})$ *then* $\theta\Gamma \vdash \mathrm{t} : \mathrm{T}$

S is a *principal type scheme* of t under Γ iff

1. $\Gamma \vdash \mathrm{t} : \mathrm{S}$
2. for every other S' such that $\Gamma \vdash \mathrm{t} : \mathrm{S}$ we have $\mathrm{S} \sqsubseteq \mathrm{S}'$

Theorem (Completeness of $\mathcal{W}$)

If $\Gamma \vdash \mathrm{t} : \mathrm{T}$ *for some* T *then* $\mathcal{W}(\Gamma, \mathrm{t}) = (\theta, \mathrm{T}')$*, and, moreover* $\mathrm{T}' \sqsubseteq \mathrm{T}$ *(i.e.* $\mathcal{W}$ *finds a principal type scheme for* t *under* Γ*)*

Algorithm W

Time complexity?

- proven to be *DEXPTIME-hard*
- *non-linear behaviour* manifests only on pathological inputs
- *polynomial* if depth of `let` nesting is bounded

Alternative approach to type inference

- Let's look at some terms:
  - $\mathrm{t}_1\ \mathrm{t}_2$: for this to typecheck, t_1 must be of type $\alpha \to \beta$, t_2 of type α and $\mathrm{t}_1\ \mathrm{t}_2$ of type β
  - $\mathrm{t}_1 + \mathrm{t}_2$: for this to typecheck, t_1 must be of type Nat, t_2 of type Nat and $\mathrm{t}_1 + \mathrm{t}_2$ of type Nat
- ...these are constraints!
- If we solve the constraint system, we have our typing.

Constraint generation

- label each term with a new type variable
- generate constraints according to rules above
- example 1: $\mathrm{t}_1\ \mathrm{t}_2$
  - type variables: $\mathrm{t}_1 : \alpha_1, \mathrm{t}_2 : \alpha_2, \mathrm{t}_1\ \mathrm{t}_2 : \beta$
  - constraints: $\alpha_1 = \alpha_2 \to \beta$
- example 2: $\mathrm{t}_1 + \mathrm{t}_2$
  - type variables: $\mathrm{t}_1 : \alpha_1, \mathrm{t}_2 : \alpha_2, \mathrm{t}_1\ +\ \mathrm{t}_2 : \beta$
  - constraints: $\alpha_1 = \mathrm{Nat}, \alpha_2 = \mathrm{Nat}, \beta = \mathrm{Nat}$

Constraint generation

Select rules

term	type	constraints
$1, 2, 3, \ldots$	Nat	
false	Bool	
nil	$\mathrm{List}\ \alpha$	
$\mathrm{t}_1\ \mathrm{t}_2$	β	$\mathrm{t}_1 : \alpha \to \beta, t_2 : \alpha$
$\lambda x.\mathrm{t}$	$\alpha \to \beta$	$x : \alpha, \mathrm{t} : \beta$
$\mathrm{t}_1 + \mathrm{t}_2$	Nat	$\mathrm{t}_1 : \mathrm{Nat}, \mathrm{t}_2 : \mathrm{Nat}$
$\mathrm{t}_1 * \mathrm{t}_2$	Nat	$\mathrm{t}_1 : \mathrm{Nat}, \mathrm{t}_2 : \mathrm{Nat}$
$\mathrm{if}\ \ \mathrm{t}_1\ \ \mathrm{then}\ \ \mathrm{t}_2\ \ \mathrm{else}\ \ \mathrm{t}_3$	α	$\mathrm{t}_1 : \mathrm{Bool}, \mathrm{t}_2 : \alpha, \mathrm{t}_3 : \alpha$
$\mathrm{hd}\ \ \mathrm{t}$	α	$\mathrm{t} : \mathrm{List}\ \alpha$
$\mathrm{tl}\ \ \mathrm{t}$	$\mathrm{List}\ \alpha$	$\mathrm{t} : \mathrm{List}\ \alpha$
$\mathrm{cons}\ \ \mathrm{t}_1\ \mathrm{t}_2$	$\mathrm{List}\ \alpha$	$\mathrm{t}_1 : \alpha, \mathrm{t}_2 : \mathrm{List}\ \alpha$

Polymorphism II – Beyond HM

Typing recursion in HM

Problem: $\boldsymbol{Y}$ is not typeable in HM

Solution:

- add fix of type $\forall\alpha.(\alpha \to \alpha) \to \alpha$
- together with the appropriate typing and evaluation rules
- `let rec` is then defined using `let` and fix
- see extensions to $\lambda^{\to}$(Lecture 3)

Other terms not typeable in HM

What is the type of $\lambda f.(f\,\mathrm{true}, f\ 0)$?

- $(\forall\alpha.\alpha \to \alpha) \to (\mathrm{Bool}, \mathrm{Nat})$?
- $(\forall\alpha.\alpha \to \mathrm{Nat}) \to (\mathrm{Nat}, \mathrm{Nat})$?
- $(\forall\alpha.\alpha \to \beta) \to (\beta, \beta)$?

In all cases, the argument is a “normal” polymorphic function.

Church numerals

- $\underline{0} := \lambda f.\lambda x.x$
- $\underline{1} := \lambda f.\lambda x.f\ x$
- …
- $\underline{n} := \lambda f.\lambda x.f^n(x)$

Type of $\underline{n}$? $\forall\alpha : (\alpha \to \alpha) \to \alpha \to \alpha$
Now try to type succ $:= \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$!

Type rank

Rank

- Universally quantified types have *rank 1*.
- Functions of *rank n* have at least one argument of rank n-1, but no arguments of a higher rank.

examples

$\forall\alpha.\alpha \to \alpha$	rank 1
$(\forall\alpha.\alpha \to \alpha) \to \mathrm{Nat}$	rank 2
$\mathrm{Nat} \to (\forall\alpha.\alpha \to \alpha) \to \mathrm{Nat} \to \mathrm{Nat}$	rank 2
$((\forall\alpha.\alpha \to \alpha) \to \mathrm{Nat}) \to \mathrm{Nat}$	rank 3

Note: System HM: only rank 1 types!

System F – syntax

term and values

$t ::=$	x	variable
$\mid$	$t\ t'$	application
$\mid$	$\lambda x : T.t$	abstraction
$\mid$	$\Lambda\alpha.t$	type abstraction
$\mid$	$t\ [T]$	type application

$v ::=$	$\lambda x : T.t$	abstraction value
$\mid$	$\Lambda\alpha.t$	type abstraction value

types

$T ::=$	α	type *variable*
$\mid$	$T \to T$	function type
$\mid$	$\forall\vec{\alpha}.T$	universal type

System F – typing and evaluation

We need to extend $\lambda^{\rightarrow}$ with the following rules:

typing

$$\frac{\Gamma \vdash \mathrm{t} : \mathrm{T} \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash \Lambda\alpha.\mathrm{t} : \forall\alpha.\mathrm{T}}$$ (T-TAbs) *(T-Gen)*

$$\frac{\Gamma \vdash \mathrm{t} : \forall\alpha.\mathrm{T}_1}{\Gamma \vdash \mathrm{t}\ [\mathrm{T}_2] : \{\mathrm{T}_2/\alpha\}\mathrm{T}_1}$$ (T-TApp) *(T-Inst)*

evaluation

$$\frac{\mathrm{t} \rightarrow \mathrm{t}'}{\mathrm{t}\ [\mathrm{T}] \rightarrow \mathrm{t}'\ [\mathrm{T}]}$$ (E-Tapp)

$(\Lambda\alpha.\mathrm{t})\ [\mathrm{T}] \rightarrow \{\mathrm{T}/\alpha\}\mathrm{t}$ (E-TappTabs)

System F – examples

Identity

$$\mathrm{id} = \Lambda\alpha.\lambda x : \alpha.x \qquad \mathrm{id} : \forall\alpha.\alpha \to \alpha$$

$$\mathrm{id}\ [\mathrm{Nat}] : \mathrm{Nat} \to \mathrm{Nat} \qquad (\mathrm{id}\ [\mathrm{Nat}]\ 0) \to 0$$

$$\mathrm{id}\ [\mathrm{Bool}] : \mathrm{Bool} \to \mathrm{Bool}$$

$$\mathrm{double} = \Lambda\alpha.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f\ (f\ x)$$

$$\mathrm{double} : \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$$

$$\mathrm{double}\ [\mathrm{Nat}] : (\mathrm{Nat} \to \mathrm{Nat}) \to \mathrm{Nat} \to \mathrm{Nat}$$

$$\mathrm{quadruple} = \Lambda\alpha.\,\mathrm{double}\ [\alpha \to \alpha]\ (\mathrm{double}\ [\alpha])$$

$$\mathrm{quadruple} : \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$$

Typing self-application and recursion

Recall: $\omega = \lambda x.x\ x$ is *not typeable* in $\lambda^{\rightarrow}$

$$\text{self} = \lambda x : \forall\alpha.\alpha \rightarrow \alpha.x\ [\forall\alpha.\alpha \rightarrow \alpha]\ x$$
$$\text{self} : (\forall\alpha.\alpha \rightarrow \alpha) \rightarrow (\forall\alpha.\alpha \rightarrow \alpha)$$

Evaluation

$$\begin{aligned} \text{self}\ \ \text{id} \rightarrow \text{id}\ \ [\forall\alpha.\alpha \rightarrow \alpha]\ \ \text{id} = \\ (\Lambda\beta.\lambda y : \beta.y)\ [\forall\alpha.\alpha \rightarrow \alpha]\ \ \text{id} \rightarrow \\ (\lambda y : (\forall\alpha.\alpha \rightarrow \alpha).y)\ \ \text{id} \rightarrow \\ \text{id} \end{aligned}$$

Typing $\boldsymbol{Y}$ / $\mathbf{fix}$

Easy peasy: $\text{fix} : \forall\alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha$!

System F – Type safety

Theorem (Progress)

Let t *be a closed, well-typed term (i.e.* $\exists \mathrm{T}$ *s.t.* $\vdash \mathrm{t} : \mathrm{T}$*). Then either* t *is a value, or there exists* t' *such that* $\mathrm{t} \to \mathrm{t}'$*.*

Theorem (Preservation)

If $\Gamma \vdash \mathrm{t} : \mathrm{T}$ *and* $\mathrm{t} \to \mathrm{t}'$*, then* $\Gamma \vdash \mathrm{t}' : \mathrm{T}$

Proofs of both theorems are simple extensions of the corresponding proofs for $\lambda^{\to}$.

System F – Normalization

Theorem (Normalization)

Well-typed System F terms are (strongly) normalizing.

Proof.

Actually quite difficult and surprising [Girard 1972]. □

System F – Type inference

Type erasing

$$\begin{aligned} erase(x) &= x \\ erase(\lambda x:\mathrm{T}.\mathrm{t}) &= \lambda x.\, erase(\mathrm{t}) \\ erase(\mathrm{t}_1\ \mathrm{t}_2) &= erase(\mathrm{t}_1)\ \ erase(\mathrm{t}_2) \\ erase(\Lambda\alpha.\mathrm{t}) &= erase(\mathrm{t}) \\ erase(\mathrm{t}\ [\mathrm{T}]) &= erase(\mathrm{t}) \end{aligned}$$

Theorem (Wells 1994)

It is undecidable whether, given a closed term M of the untyped lambda-calculus, there is some well-typed term t in System F s.t. $erase(\mathrm{t}) = M$.

Note: *Type-checking* is decidable.

System F – type inference 2

Theorem (Kfoury, Wells 1999)

Type reconstruction for rank ≥ 3 *is undecidable.*

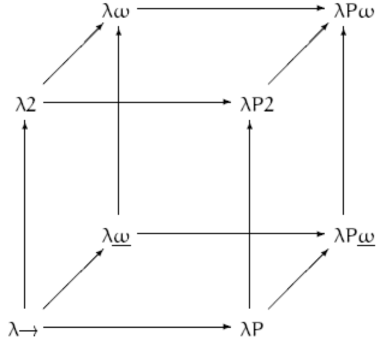
Practical solutions

- limit your language to rank 1 types
  - approach of ML and HASKELL98
- limit your language to rank 2 types
  - the complexity of type inference is the same as for HINDLEY-MILNER
- use System F and *provide some type information*
  - approach of HASKELL

Beyond System F

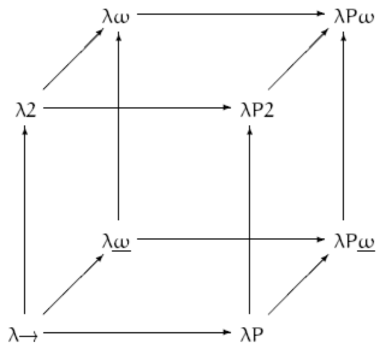
Lambda cube

- classification of type systems
- starts with $\lambda^{\rightarrow}$
- extensions:
  1. polymorphic types
  2. type operations
  3. dependent types
- $\lambda 2$ – System F



Dependencies between types and terms

- terms depending on terms
  normal functions
- terms depending on types
  polymorphism
- types depending on types
  type operators
- types depending on terms
  dependent types



Corners of the Lambda cube

$\lambda^{\rightarrow}$ – simply typed lambda calculus

$\lambda 2$ – polymorphic lambda calculus (System F)

$\lambda \underline{\omega}$ – simply typed lambda calculus with type operators

$\lambda \omega$ – System F_{ω}

$\lambda P \omega$ – calculus of constructions

λP – dependent types (LF)

All eight are strongly normalizing!